Benchmarking benchmarking, and optimizing optimization

Daniel J. Bernstein

University of Illinois at Chicago &
Technische Universiteit Eindhoven

Bit operations per bit of plaintext (assuming precomputed subkeys), as listed in recent Skinny paper:

key	ops/bit	cipher
128	88	Simon: 60 ops broken
128	100	NOEKEON
128	117	Skinny
256	144	Simon: 106 ops broken
128	147.2	PRESENT
256	156	Skinny
128	162.75	Piccolo
128	202.5	AES
256	283.5	AES

Benchmarking benchmarking, and optimizing optimization

Daniel J. Bernstein

University of Illinois at Chicago &
Technische Universiteit Eindhoven

Bit operations per bit of plaintext (assuming precomputed subkeys), not entirely listed in Skinny paper:

key	ops/bit	cipher
256	54	Salsa20/8
256	78	Salsa20/12
128	88	Simon: 60 ops broken
128	100	NOEKEON
128	117	Skinny
256	126	Salsa20
256	144	Simon: 106 ops broken
128	147.2	PRESENT
256	156	Skinny
128	162.75	Piccolo
128	202.5	AES
256	283.5	AES

arking benchmarking,
mizing optimization

. Bernstein

ty of Illinois at Chicago &
che Universiteit Eindhoven

Bit operations per bit of plaintext (assuming precomputed subkeys), not entirely listed in Skinny paper:

key	ops/bit	cipher
256	54	Salsa20/8
256	78	Salsa20/12
128	88	Simon: 60 ops broken
128	100	NOEKEON
128	117	Skinny
256	126	Salsa20
256	144	Simon: 106 ops broken
128	147.2	PRESENT
256	156	Skinny
128	162.75	Piccolo
128	202.5	AES
256	283.5	AES

Operatio
poor mo
worse m

Pick a c
How fas

First ste
Write si

e.g. Ber
Janssen-
Smetsers
includes
impleme

nchmarking,
timization

n

is at Chicago &
siteit Eindhoven

Bit operations per bit of plaintext (assuming precomputed subkeys), not entirely listed in Skinny paper:

key	ops/bit	cipher
256	54	Salsa20/8
256	78	Salsa20/12
128	88	Simon: 60 ops broken
128	100	NOEKEON
128	117	Skinny
256	126	Salsa20
256	144	Simon: 106 ops broken
128	147.2	PRESENT
256	156	Skinny
128	162.75	Piccolo
128	202.5	AES
256	283.5	AES

Operation counts
poor model of har
worse model of so

Pick a cipher: e.g.
How fast is Salsa2

First step in analy
Write simple softw

e.g. Bernstein–va
Janssen–Lange–Sc
Smetsers "TweetN
includes essentially
implementation of

g,

ago &

hoven

Bit operations per bit of plaintext (assuming precomputed subkeys), not entirely listed in Skinny paper:

key	ops/bit	cipher
256	54	Salsa20/8
256	78	Salsa20/12
128	88	Simon: 60 ops broken
128	100	NOEKEON
128	117	Skinny
256	126	Salsa20
256	144	Simon: 106 ops broken
128	147.2	PRESENT
256	156	Skinny
128	162.75	Piccolo
128	202.5	AES
256	283.5	AES

Operation counts are a
poor model of hardware cos
worse model of software cos

Pick a cipher: e.g., Salsa20.
How fast is Salsa20 softwar

First step in analysis:
Write simple software.

e.g. Bernstein–van Gastel–
Janssen–Lange–Schwabe–
Smetsers "TweetNaCl"
includes essentially the follo
implementation of Salsa20:

Bit operations per bit of plaintext (assuming precomputed subkeys), not entirely listed in Skinny paper:

key	ops/bit	cipher
256	54	Salsa20/8
256	78	Salsa20/12
128	88	Simon: 60 ops broken
128	100	NOEKEON
128	117	Skinny
256	126	Salsa20
256	144	Simon: 106 ops broken
128	147.2	PRESENT
256	156	Skinny
128	162.75	Piccolo
128	202.5	AES
256	283.5	AES

Operation counts are a poor model of hardware cost, worse model of software cost.

Pick a cipher: e.g., Salsa20. How fast is Salsa20 software?

First step in analysis: Write simple software.

e.g. Bernstein–van Gastel–Janssen–Lange–Schwabe–Smetsers “TweetNaCl” includes essentially the following implementation of Salsa20:

ations per bit of plaintext
ng precomputed subkeys),
rely listed in Skinny paper:

s/bit	cipher
4	Salsa20/8
3	Salsa20/12
3	Simon: 60 ops broken
0	NOEKEON
7	Skinny
6	Salsa20
4	Simon: 106 ops broken
7.2	PRESENT
6	Skinny
2.75	Piccolo
2.5	AES
3.5	AES

Operation counts are a poor model of hardware cost, worse model of software cost.

Pick a cipher: e.g., Salsa20. How fast is Salsa20 software?

First step in analysis: Write simple software.

e.g. Bernstein–van Gastel–Janssen–Lange–Schwabe–Smetsers “TweetNaCl” includes essentially the following implementation of Salsa20:

```
int crypt
const u8
{
  u32 w[1
  int i,j


  FOR(i,4
    x[5*i
    x[1+i
    x[6+i
    x[11+
  }


  FOR(i,1
```

bit of plaintext
puted subkeys),
in Skinny paper:

er

---

a20/8
a20/12
on: 60 ops broken
EKEON
ny
a20
on: 106 ops broken
SENT
ny
olo

Operation counts are a poor model of hardware cost, worse model of software cost.

Pick a cipher: e.g., Salsa20. How fast is Salsa20 software?

First step in analysis: Write simple software.

e.g. Bernstein–van Gastel–Janssen–Lange–Schwabe–Smetsers “TweetNaCl” includes essentially the following implementation of Salsa20:

```
int crypto_core_salsa
const u8 *in,const u8
{
  u32 w[16],x[16],y[1
  int i,j,m;

  FOR(i,4) {
    x[5*i] = ld32(c+4
    x[1+i] = ld32(k+4
    x[6+i] = ld32(in+
    x[11+i] = ld32(k+
  }

  FOR(i,16) y[i] = x[
```

intext
keys),
paper:

---

broken

s broken

Operation counts are a poor model of hardware cost, worse model of software cost.

Pick a cipher: e.g., Salsa20. How fast is Salsa20 software?

First step in analysis: Write simple software.

e.g. Bernstein–van Gastel–Janssen–Lange–Schwabe–Smetsers “TweetNaCl” includes essentially the following implementation of Salsa20:

```
int crypto_core_salsa20(u8 *out,
const u8 *in,const u8 *k,const u
{
  u32 w[16],x[16],y[16],t[4];
  int i,j,m;

  FOR(i,4) {
    x[5*i] = ld32(c+4*i);
    x[1+i] = ld32(k+4*i);
    x[6+i] = ld32(in+4*i);
    x[11+i] = ld32(k+16+4*i);
  }

  FOR(i,16) y[i] = x[i];
```

Operation counts are a poor model of hardware cost, worse model of software cost.

Pick a cipher: e.g., Salsa20. How fast is Salsa20 software?

First step in analysis: Write simple software.

e.g. Bernstein–van Gastel–Janssen–Lange–Schwabe–Smetsers “TweetNaCl” includes essentially the following implementation of Salsa20:

```
int crypto_core_salsa20(u8 *out,
const u8 *in,const u8 *k,const u8 *c)
{
  u32 w[16],x[16],y[16],t[4];
  int i,j,m;

  FOR(i,4) {
    x[5*i] = ld32(c+4*i);
    x[1+i] = ld32(k+4*i);
    x[6+i] = ld32(in+4*i);
    x[11+i] = ld32(k+16+4*i);
  }

  FOR(i,16) y[i] = x[i];
```

on counts are a
odel of hardware cost,
odel of software cost.

ipher: e.g., Salsa20.
t is Salsa20 software?

p in analysis:
mple software.

nstein–van Gastel–
-Lange–Schwabe–
s "TweetNaCl"
essentially the following
ntation of Salsa20:

```
int crypto_core_salsa20(u8 *out,
const u8 *in,const u8 *k,const u8 *c)
{
  u32 w[16],x[16],y[16],t[4];
  int i,j,m;

  FOR(i,4) {
    x[5*i] = ld32(c+4*i);
    x[1+i] = ld32(k+4*i);
    x[6+i] = ld32(in+4*i);
    x[11+i] = ld32(k+16+4*i);
  }

  FOR(i,16) y[i] = x[i];
```

```
  FOR(i,20
    FOR(j
      FOR
      t[1
      t[2
      t[3
      t[0
      FOR
    }
    FOR(m
  }

  FOR(i,16
  return 
}
```

are a
dware cost,
ftware cost.

., Salsa20.
0 software?

sis:
are.

n Gastel–
chwabe–
aCl”
the following
Salsa20:

```
int crypto_core_salsa20(u8 *out,
const u8 *in,const u8 *k,const u8 *c)
{
  u32 w[16],x[16],y[16],t[4];
  int i,j,m;

  FOR(i,4) {
    x[5*i] = ld32(c+4*i);
    x[1+i] = ld32(k+4*i);
    x[6+i] = ld32(in+4*i);
    x[11+i] = ld32(k+16+4*i);
  }

  FOR(i,16) y[i] = x[i];
```

```
  FOR(i,20) {
    FOR(j,4) {
      FOR(m,4) t[m] =
      t[1] ^= L32(t[0
      t[2] ^= L32(t[1
      t[3] ^= L32(t[2
      t[0] ^= L32(t[3
      FOR(m,4) w[4*j+
    }
    FOR(m,16) x[m] =
  }

  FOR(i,16) st32(out
  return 0;
}
```

t,

t.

e?

wing

```
int crypto_core_salsa20(u8 *out,
const u8 *in,const u8 *k,const u8 *c)
{
  u32 w[16],x[16],y[16],t[4];
  int i,j,m;

  FOR(i,4) {
    x[5*i] = ld32(c+4*i);
    x[1+i] = ld32(k+4*i);
    x[6+i] = ld32(in+4*i);
    x[11+i] = ld32(k+16+4*i);
  }

  FOR(i,16) y[i] = x[i];
```

```
  FOR(i,20) {
    FOR(j,4) {
      FOR(m,4) t[m] = x[(5*j+4*m
      t[1] ^= L32(t[0]+t[3], 7);
      t[2] ^= L32(t[1]+t[0], 9);
      t[3] ^= L32(t[2]+t[1],13);
      t[0] ^= L32(t[3]+t[2],18);
      FOR(m,4) w[4*j+(j+m)%4] =
    }
    FOR(m,16) x[m] = w[m];
  }

  FOR(i,16) st32(out + 4 * i,x[i
  return 0;
}
```

```
int crypto_core_salsa20(u8 *out,
const u8 *in,const u8 *k,const u8 *c)
{
  u32 w[16],x[16],y[16],t[4];
  int i,j,m;

  FOR(i,4) {
    x[5*i] = ld32(c+4*i);
    x[1+i] = ld32(k+4*i);
    x[6+i] = ld32(in+4*i);
    x[11+i] = ld32(k+16+4*i);
  }

  FOR(i,16) y[i] = x[i];
```

```
  FOR(i,20) {
    FOR(j,4) {
      FOR(m,4) t[m] = x[(5*j+4*m)%16];
      t[1] ^= L32(t[0]+t[3], 7);
      t[2] ^= L32(t[1]+t[0], 9);
      t[3] ^= L32(t[2]+t[1],13);
      t[0] ^= L32(t[3]+t[2],18);
      FOR(m,4) w[4*j+(j+m)%4] = t[m];
    }
    FOR(m,16) x[m] = w[m];
  }

  FOR(i,16) st32(out + 4 * i,x[i] + y[i]);
  return 0;
}
```

```
o_core_salsa20(u8 *out,
*in,const u8 *k,const u8 *c)

6],x[16],y[16],t[4];
,m;

) {
] = ld32(c+4*i);
] = ld32(k+4*i);
] = ld32(in+4*i);
i] = ld32(k+16+4*i);


6) y[i] = x[i];
```

```
  FOR(i,20) {
    FOR(j,4) {
      FOR(m,4) t[m] = x[(5*j+4*m)%16];
      t[1] ^= L32(t[0]+t[3], 7);
      t[2] ^= L32(t[1]+t[0], 9);
      t[3] ^= L32(t[2]+t[1],13);
      t[0] ^= L32(t[3]+t[2],18);
      FOR(m,4) w[4*j+(j+m)%4] = t[m];
    }
    FOR(m,16) x[m] = w[m];
  }

  FOR(i,16) st32(out + 4 * i,x[i] + y[i]);
  return 0;
}
```

```
static co
= "expand

int crypt
const u8
{
  u8 z[16
  u32 u,i
  if (!b)
  FOR(i,1
  FOR(i,8
  while (
    crypt
    FOR(i
    u = 1
```

```
a20(u8 *out,
3 *k,const u8 *c)

.6],t[4];



4*i);
4*i);
-4*i);
-16+4*i);


[i];
```

```
  FOR(i,20) {
    FOR(j,4) {
      FOR(m,4) t[m] = x[(5*j+4*m)%16];
      t[1] ^= L32(t[0]+t[3], 7);
      t[2] ^= L32(t[1]+t[0], 9);
      t[3] ^= L32(t[2]+t[1],13);
      t[0] ^= L32(t[3]+t[2],18);
      FOR(m,4) w[4*j+(j+m)%4] = t[m];
    }
    FOR(m,16) x[m] = w[m];
  }

  FOR(i,16) st32(out + 4 * i,x[i] + y[i]);
  return 0;
}
```

```
static const u8 sigma
= "expand 32-byte k";

int crypto_stream_sal
const u8 *m,u64 b,con
{
  u8 z[16],x[64];
  u32 u,i;
  if (!b) return 0;
  FOR(i,16) z[i] = 0;
  FOR(i,8) z[i] = n[i
  while (b >= 64) {
    crypto_core_salsa
    FOR(i,64) c[i] =
    u = 1;
```

```
8 *c)
```

```
  FOR(i,20) {
    FOR(j,4) {
      FOR(m,4) t[m] = x[(5*j+4*m)%16];
      t[1] ^= L32(t[0]+t[3], 7);
      t[2] ^= L32(t[1]+t[0], 9);
      t[3] ^= L32(t[2]+t[1],13);
      t[0] ^= L32(t[3]+t[2],18);
      FOR(m,4) w[4*j+(j+m)%4] = t[m];
    }
    FOR(m,16) x[m] = w[m];
  }

  FOR(i,16) st32(out + 4 * i,x[i] + y[i]);
  return 0;
}
```

```
static const u8 sigma[16]
= "expand 32-byte k";

int crypto_stream_salsa20_xor(u8
const u8 *m,u64 b,const u8 *n,co
{
  u8 z[16],x[64];
  u32 u,i;
  if (!b) return 0;
  FOR(i,16) z[i] = 0;
  FOR(i,8) z[i] = n[i];
  while (b >= 64) {
    crypto_core_salsa20(x,z,k,si
    FOR(i,64) c[i] = (m?m[i]:0)
    u = 1;
```

```
  FOR(i,20) {
    FOR(j,4) {
      FOR(m,4) t[m] = x[(5*j+4*m)%16];
      t[1] ^= L32(t[0]+t[3], 7);
      t[2] ^= L32(t[1]+t[0], 9);
      t[3] ^= L32(t[2]+t[1],13);
      t[0] ^= L32(t[3]+t[2],18);
      FOR(m,4) w[4*j+(j+m)%4] = t[m];
    }
    FOR(m,16) x[m] = w[m];
  }

  FOR(i,16) st32(out + 4 * i,x[i] + y[i]);
  return 0;
}
```

```
static const u8 sigma[16]
= "expand 32-byte k";

int crypto_stream_salsa20_xor(u8 *c,
const u8 *m,u64 b,const u8 *n,const u8 *k)
{
  u8 z[16],x[64];
  u32 u,i;
  if (!b) return 0;
  FOR(i,16) z[i] = 0;
  FOR(i,8) z[i] = n[i];
  while (b >= 64) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,64) c[i] = (m?m[i]:0) ^ x[i];
    u = 1;
```

```
0) {
,4) {
(m,4) t[m] = x[(5*j+4*m)%16];
] ^= L32(t[0]+t[3], 7);
] ^= L32(t[1]+t[0], 9);
] ^= L32(t[2]+t[1],13);
] ^= L32(t[3]+t[2],18);
(m,4) w[4*j+(j+m)%4] = t[m];

,16) x[m] = w[m];



6) st32(out + 4 * i,x[i] + y[i]);
0;
```

```
static const u8 sigma[16]
= "expand 32-byte k";

int crypto_stream_salsa20_xor(u8 *c,
const u8 *m,u64 b,const u8 *n,const u8 *k)
{
  u8 z[16],x[64];
  u32 u,i;
  if (!b) return 0;
  FOR(i,16) z[i] = 0;
  FOR(i,8) z[i] = n[i];
  while (b >= 64) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,64) c[i] = (m?m[i]:0) ^ x[i];
    u = 1;
```

```
    for (
      u +=
      z[i
      u >
    }
    b -=
    c +=
    if (m
  }
  if (b)
    crypt
    FOR(i
  }
  return
}
```

```
= x[(5*j+4*m)%16];
0]+t[3], 7);
1]+t[0], 9);
2]+t[1],13);
3]+t[2],18);
(j+m)%4] = t[m];

w[m];


+ 4 * i,x[i] + y[i]);
```

```
static const u8 sigma[16]
= "expand 32-byte k";

int crypto_stream_salsa20_xor(u8 *c,
const u8 *m,u64 b,const u8 *n,const u8 *k)
{
  u8 z[16],x[64];
  u32 u,i;
  if (!b) return 0;
  FOR(i,16) z[i] = 0;
  FOR(i,8) z[i] = n[i];
  while (b >= 64) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,64) c[i] = (m?m[i]:0) ^ x[i];
    u = 1;
```

```
    for (i = 8;i < 16
      u += (u32) z[i]
      z[i] = u;
      u >>= 8;
    }
    b -= 64;
    c += 64;
    if (m) m += 64;
  }
  if (b) {
    crypto_core_salsa
    FOR(i,b) c[i] = (
  }
  return 0;
}
```

```
)%16];



t[m];



] + y[i]);
```

```
static const u8 sigma[16]
= "expand 32-byte k";

int crypto_stream_salsa20_xor(u8 *c,
const u8 *m,u64 b,const u8 *n,const u8 *k)
{
  u8 z[16],x[64];
  u32 u,i;
  if (!b) return 0;
  FOR(i,16) z[i] = 0;
  FOR(i,8) z[i] = n[i];
  while (b >= 64) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,64) c[i] = (m?m[i]:0) ^ x[i];
    u = 1;
```

```
    for (i = 8;i < 16;++i) {
      u += (u32) z[i];
      z[i] = u;
      u >>= 8;
    }
    b -= 64;
    c += 64;
    if (m) m += 64;
  }
  if (b) {
    crypto_core_salsa20(x,z,k,si
    FOR(i,b) c[i] = (m?m[i]:0) ^
  }
  return 0;
}
```

```
static const u8 sigma[16]
= "expand 32-byte k";

int crypto_stream_salsa20_xor(u8 *c,
const u8 *m,u64 b,const u8 *n,const u8 *k)
{
  u8 z[16],x[64];
  u32 u,i;
  if (!b) return 0;
  FOR(i,16) z[i] = 0;
  FOR(i,8) z[i] = n[i];
  while (b >= 64) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,64) c[i] = (m?m[i]:0) ^ x[i];
    u = 1;
```

```
    for (i = 8;i < 16;++i) {
      u += (u32) z[i];
      z[i] = u;
      u >>= 8;
    }
    b -= 64;
    c += 64;
    if (m) m += 64;
  }
  if (b) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,b) c[i] = (m?m[i]:0) ^ x[i];
  }
  return 0;
}
```

```
nst u8 sigma[16]
 32-byte k";

o_stream_salsa20_xor(u8 *c,
*m,u64 b,const u8 *n,const u8 *k)

],x[64];
;
 return 0;
6) z[i] = 0;
) z[i] = n[i];
o >= 64) {
o_core_salsa20(x,z,k,sigma);
,64) c[i] = (m?m[i]:0) ^ x[i];
;
```

```
    for (i = 8;i < 16;++i) {
      u += (u32) z[i];
      z[i] = u;
      u >>= 8;
    }
    b -= 64;
    c += 64;
    if (m) m += 64;
  }
  if (b) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,b) c[i] = (m?m[i]:0) ^ x[i];
  }
  return 0;
}
```

Next ste

For each

compile

and see

```
a[16]

sa20_xor(u8 *c,
nst u8 *n,const u8 *k)

];

a20(x,z,k,sigma);
 (m?m[i]:0) ^ x[i];
```

```
    for (i = 8;i < 16;++i) {
      u += (u32) z[i];
      z[i] = u;
      u >>= 8;
    }
    b -= 64;
    c += 64;
    if (m) m += 64;
  }
  if (b) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,b) c[i] = (m?m[i]:0) ^ x[i];
  }
  return 0;
}
```

Next step in analy
For each target CP
compile the simple
and see how fast i

```
*c,
nst u8 *k)
gma);
^ x[i];
```

```
    for (i = 8;i < 16;++i) {
      u += (u32) z[i];
      z[i] = u;
      u >>= 8;
    }
    b -= 64;
    c += 64;
    if (m) m += 64;
  }
  if (b) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,b) c[i] = (m?m[i]:0) ^ x[i];
  }
  return 0;
}
```

Next step in analysis:
For each target CPU,
compile the simple code,
and see how fast it is.

```
    for (i = 8;i < 16;++i) {
      u += (u32) z[i];
      z[i] = u;
      u >>= 8;
    }
    b -= 64;
    c += 64;
    if (m) m += 64;
  }
  if (b) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,b) c[i] = (m?m[i]:0) ^ x[i];
  }
  return 0;
}
```

Next step in analysis:
For each target CPU,
compile the simple code,
and see how fast it is.

```
    for (i = 8;i < 16;++i) {
      u += (u32) z[i];
      z[i] = u;
      u >>= 8;
    }
    b -= 64;
    c += 64;
    if (m) m += 64;
  }
  if (b) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,b) c[i] = (m?m[i]:0) ^ x[i];
  }
  return 0;
}
```

Next step in analysis:
For each target CPU,
compile the simple code,
and see how fast it is.

In compiler writer's fantasy world,
the analysis now ends.

```
    for (i = 8;i < 16;++i) {
      u += (u32) z[i];
      z[i] = u;
      u >>= 8;
    }
    b -= 64;
    c += 64;
    if (m) m += 64;
  }
  if (b) {
    crypto_core_salsa20(x,z,k,sigma);
    FOR(i,b) c[i] = (m?m[i]:0) ^ x[i];
  }
  return 0;
}
```

Next step in analysis:
For each target CPU,
compile the simple code,
and see how fast it is.

In compiler writer's fantasy world,
the analysis now ends.

"We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

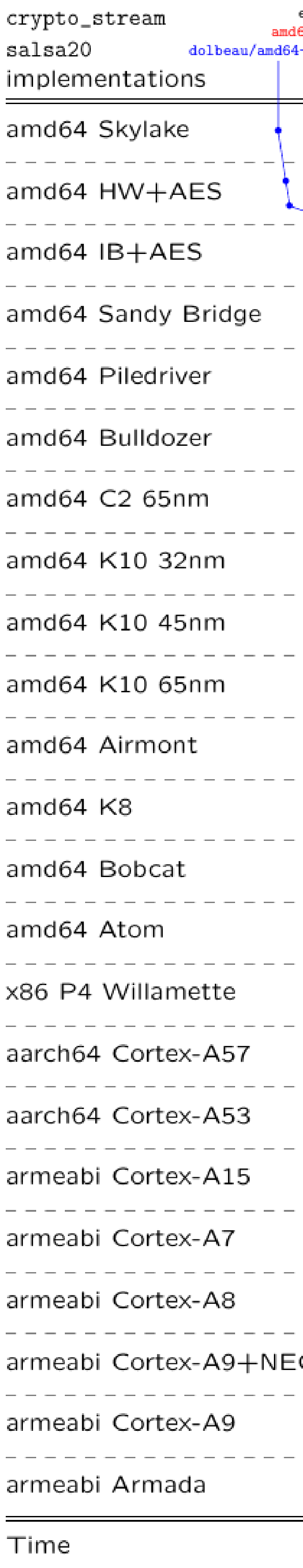
```
i = 8;i < 16;++i) {
= (u32) z[i];
] = u;
>= 8;


64;
64;
) m += 64;


{
o_core_salsa20(x,z,k,sigma);
,b) c[i] = (m?m[i]:0) ^ x[i];


0);
```

Next step in analysis:
For each target CPU,
compile the simple code,
and see how fast it is.

In compiler writer's fantasy world,
the analysis now ends.

"We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

Reality i



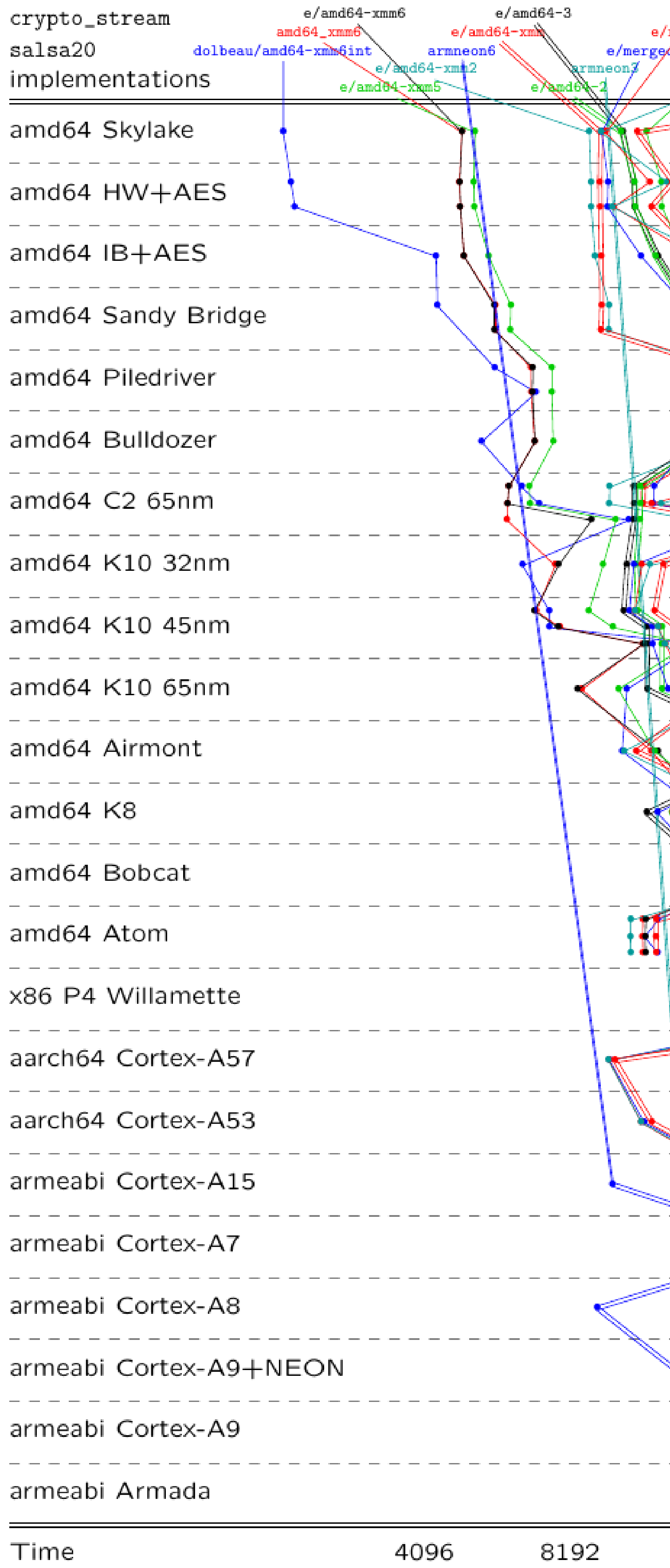
```
;++i) {

;




a20(x,z,k,sigma);

m?m[i]:0) ^ x[i];
```

Next step in analysis:
For each target CPU,
compile the simple code,
and see how fast it is.

In compiler writer's fantasy world,
the analysis now ends.

"We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

Reality is more co



gma);

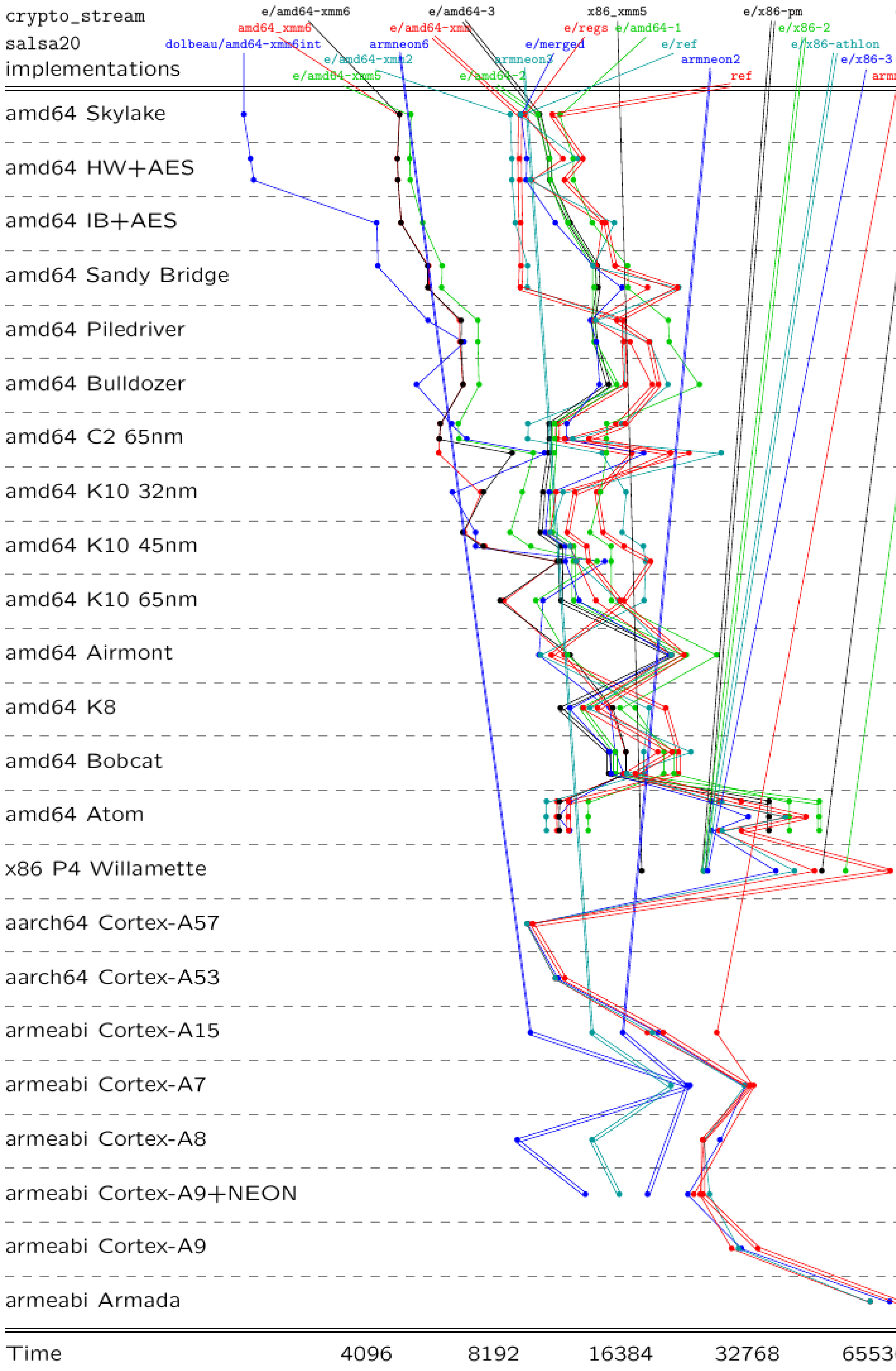
x[i];

Next step in analysis:
For each target CPU,
compile the simple code,
and see how fast it is.

In compiler writer's fantasy world,
the analysis now ends.

"We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

Reality is more complicated:

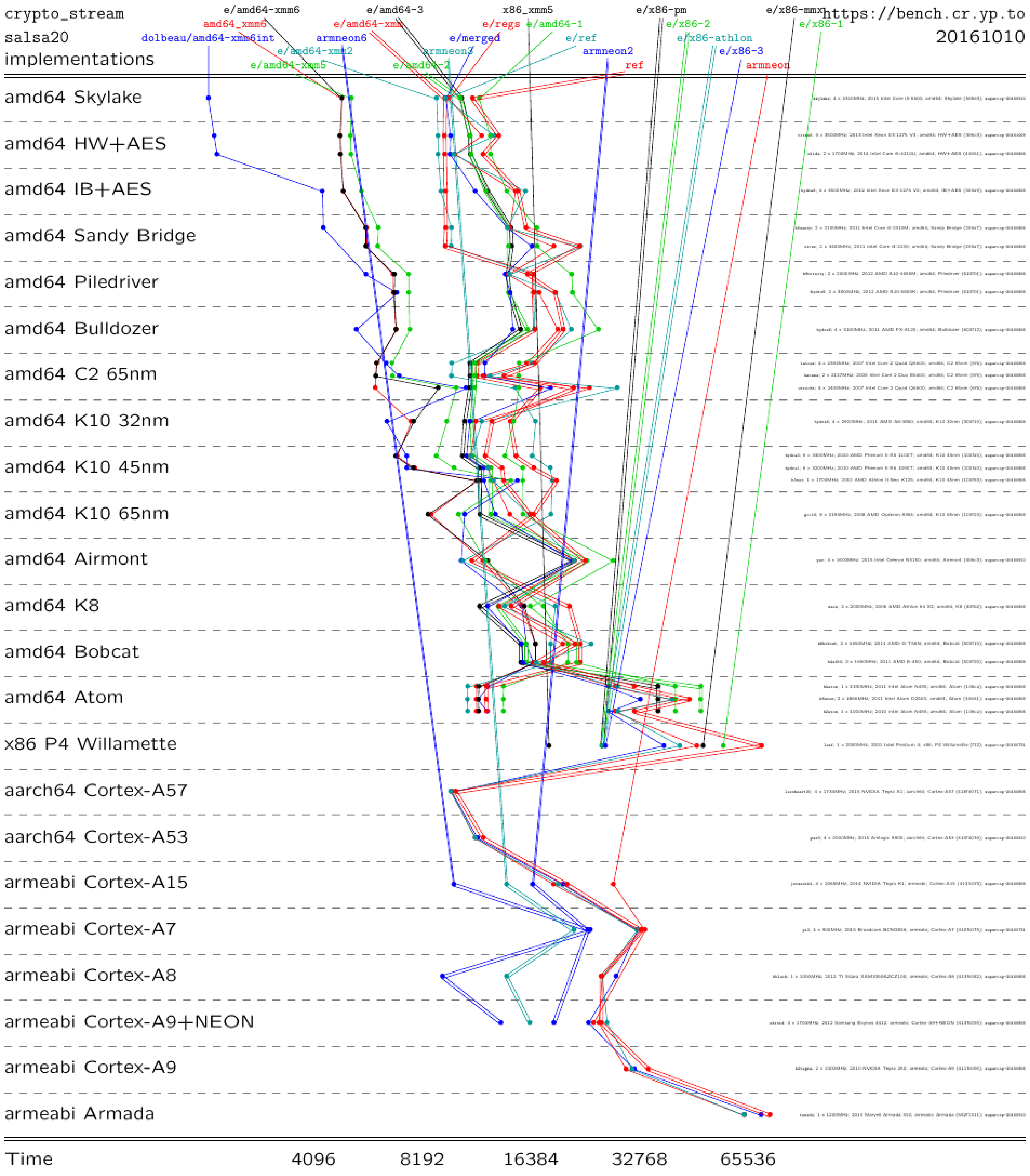


Next step in analysis:
For each target CPU,
compile the simple code,
and see how fast it is.

In compiler writer's fantasy world,
the analysis now ends.

"We come so close to optimal on most architectures that we can't do much more without using NP complete algorithms instead of heuristics. We can only try to get little niggles here and there where the heuristics get slightly wrong answers."

Reality is more complicated:

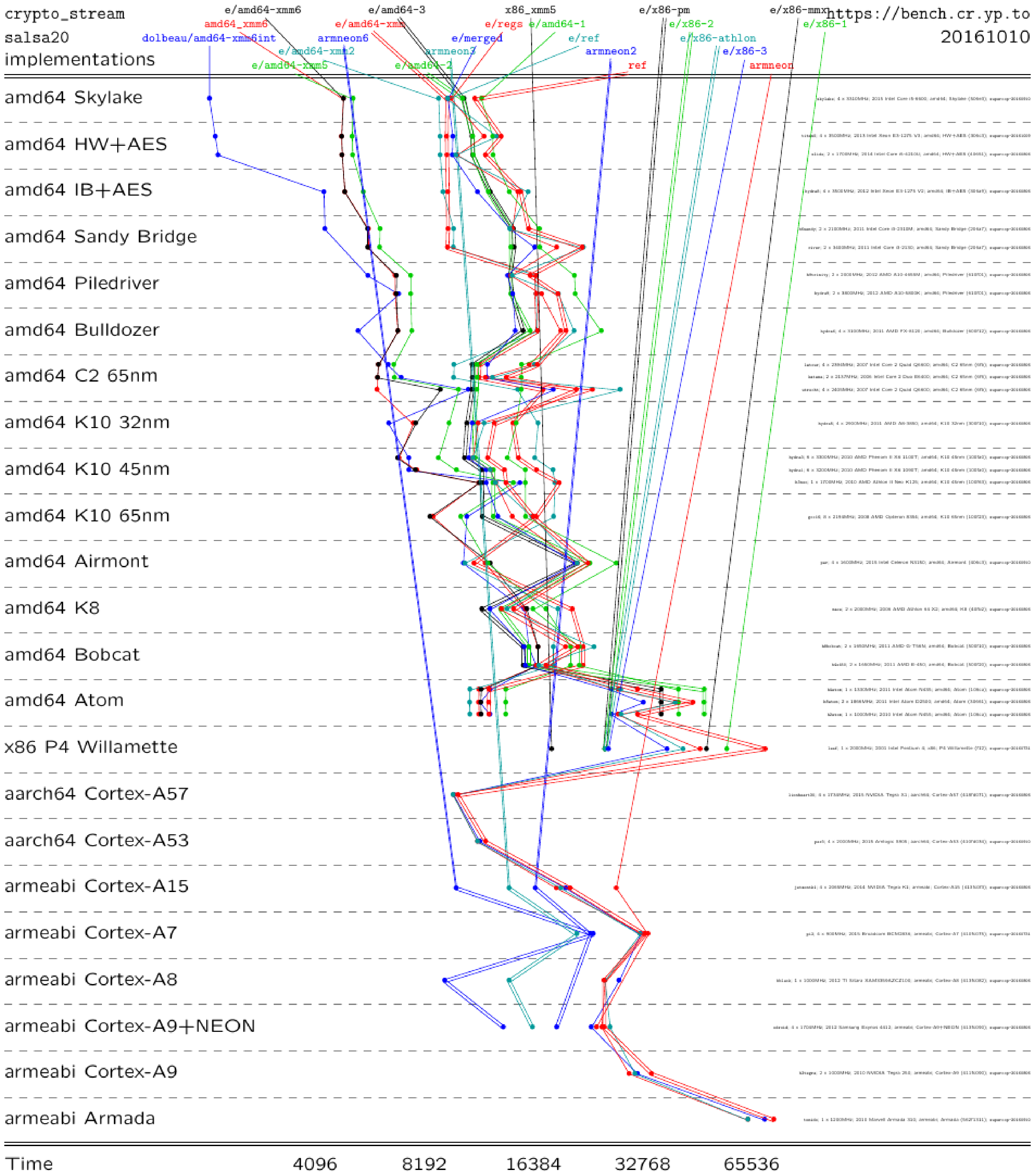


ep in analysis:
 target CPU,
the simple code,
how fast it is.

iler writer's fantasy world,
ysis now ends.

me so close to optimal on
chitectures that we can't
n more without using NP
e algorithms instead of
s. We can only try to
e niggles here and there
he heuristics get
wrong answers."

Reality is more complicated:



SUPERC
includes
of 563 c
>20 imp

Haswell:
impleme
`gcc -O3`
is 6.15×
Salsa20

`merged`
with "m
optimiza
compiler

sis:

PU,

e code,

t is.

s fantasy world,

nds.

e to optimal on

s that we can’t

hout using NP

ns instead of

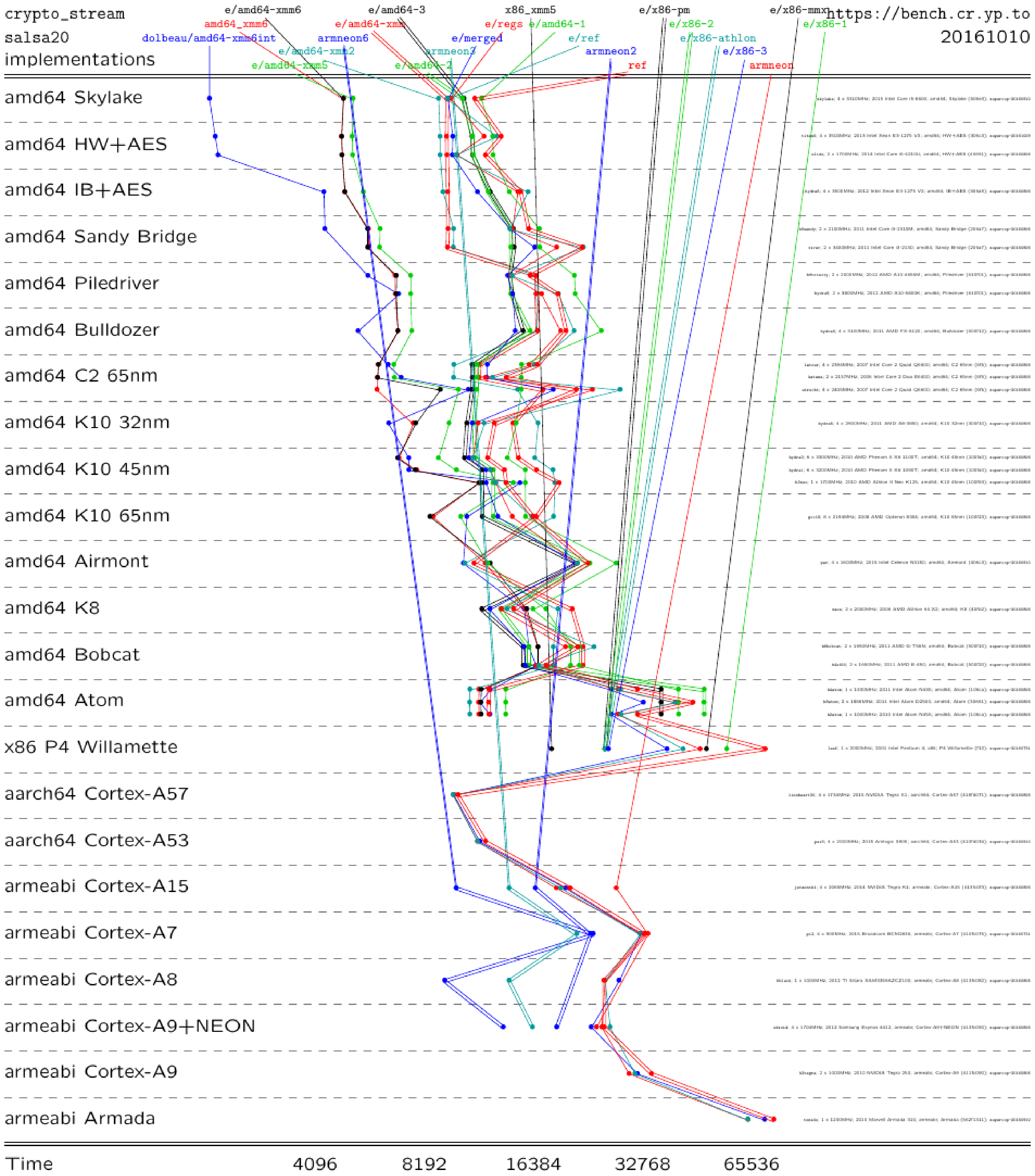
n only try to

ere and there

cs get

wers.”

Reality is more complicated:



SUPERCOP bencl

includes 2064 imp

of 563 cryptograp

>20 implementati

Haswell: Reasonal

implementation co

`gcc -O3 -fomit-`

is 6.15× slower th

Salsa20 implement

`merged` implemen

with “machine-ind

optimizations and

compiler options:

world,

al on

can’t

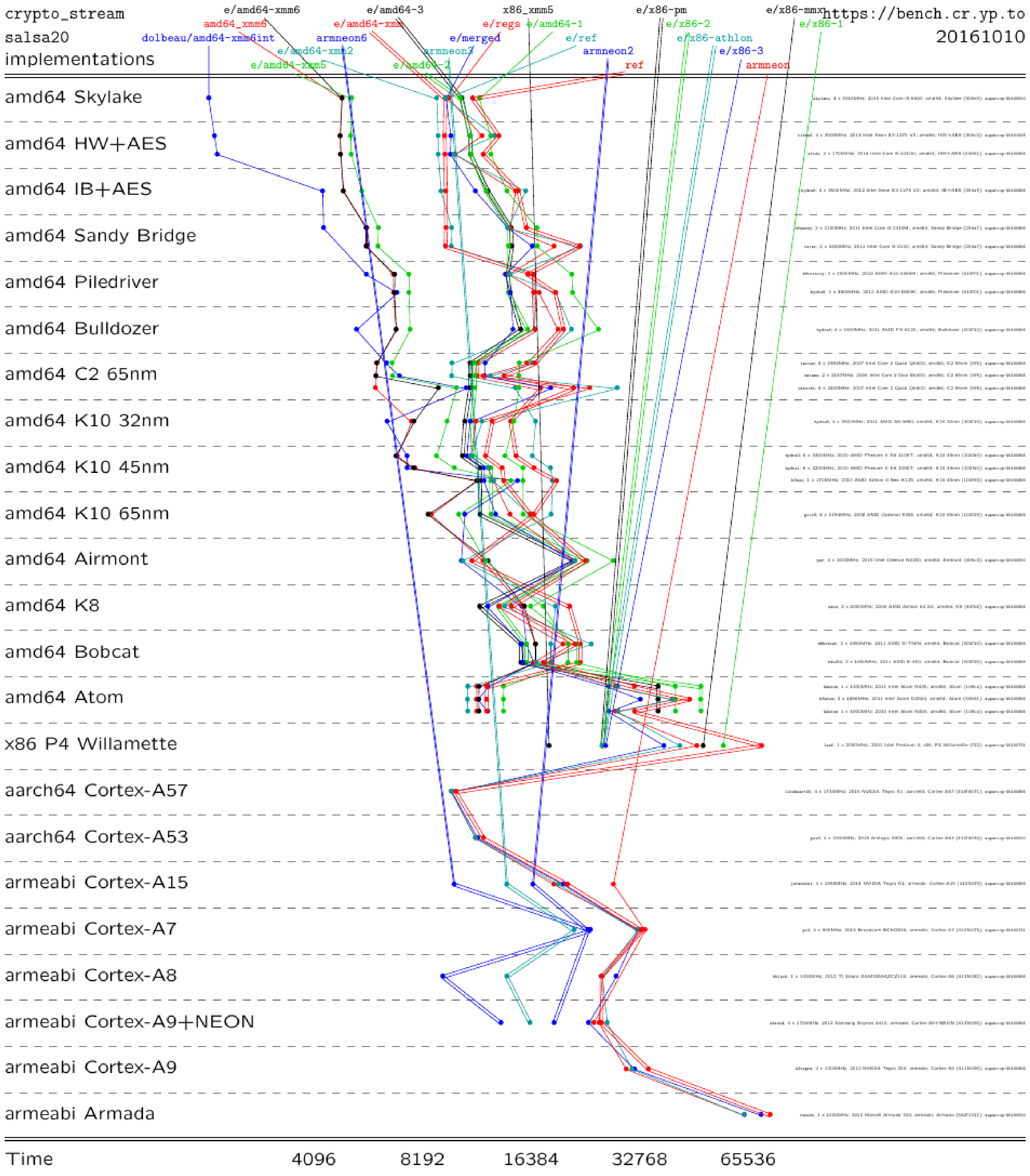
g NP

of

to

ere

Reality is more complicated:

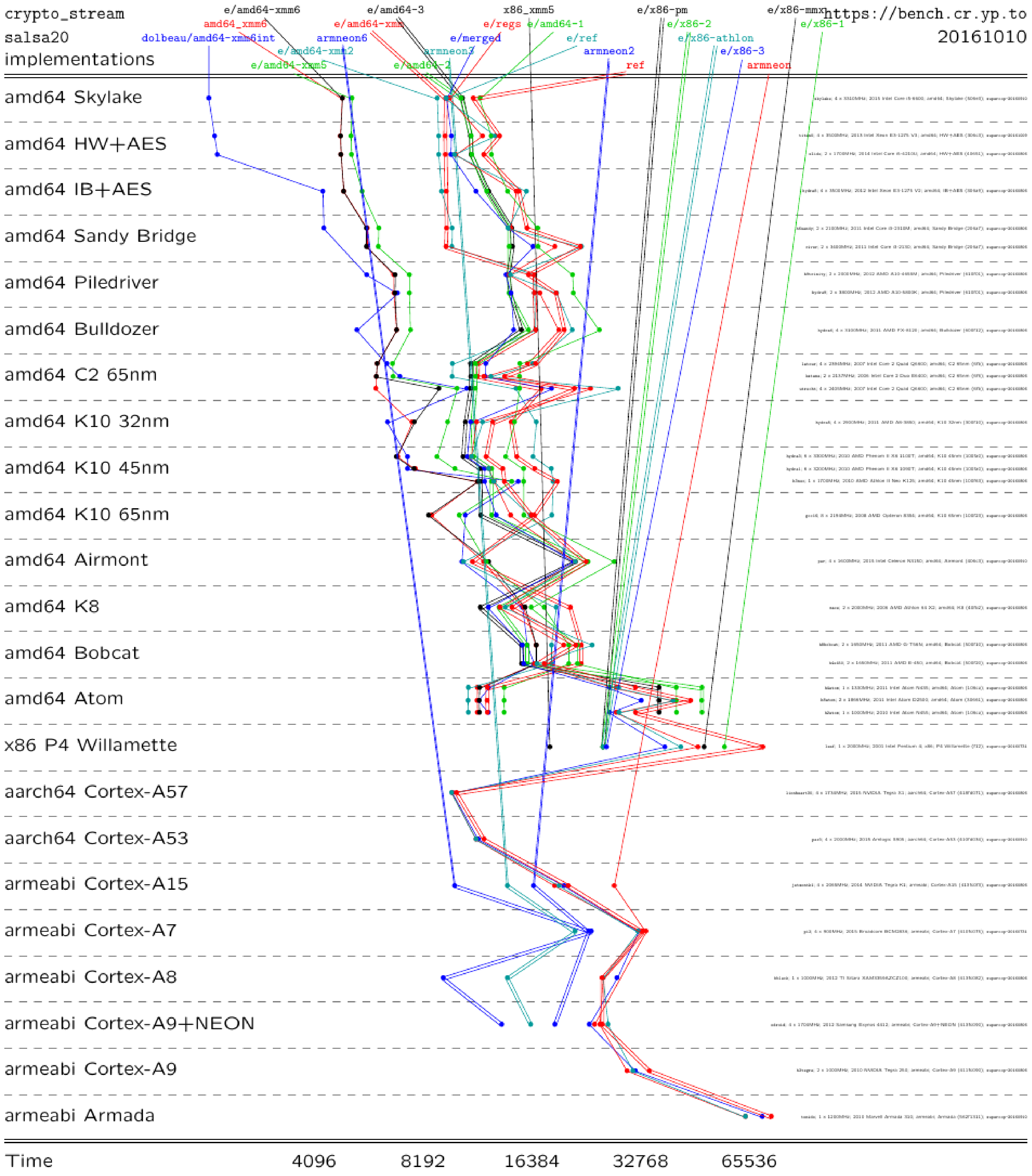


SUPERCOP benchmarking t
includes 2064 implementatio
of 563 cryptographic primiti
>20 implementations of Sal

Haswell: Reasonably simple
implementation compiled wi
`gcc -O3 -fomit-frame-po`
is 6.15× slower than fastest
Salsa20 implementation.

`merged` implementation
with “machine-independent”
optimizations and best of 12
compiler options: 4.52× slo

Reality is more complicated:

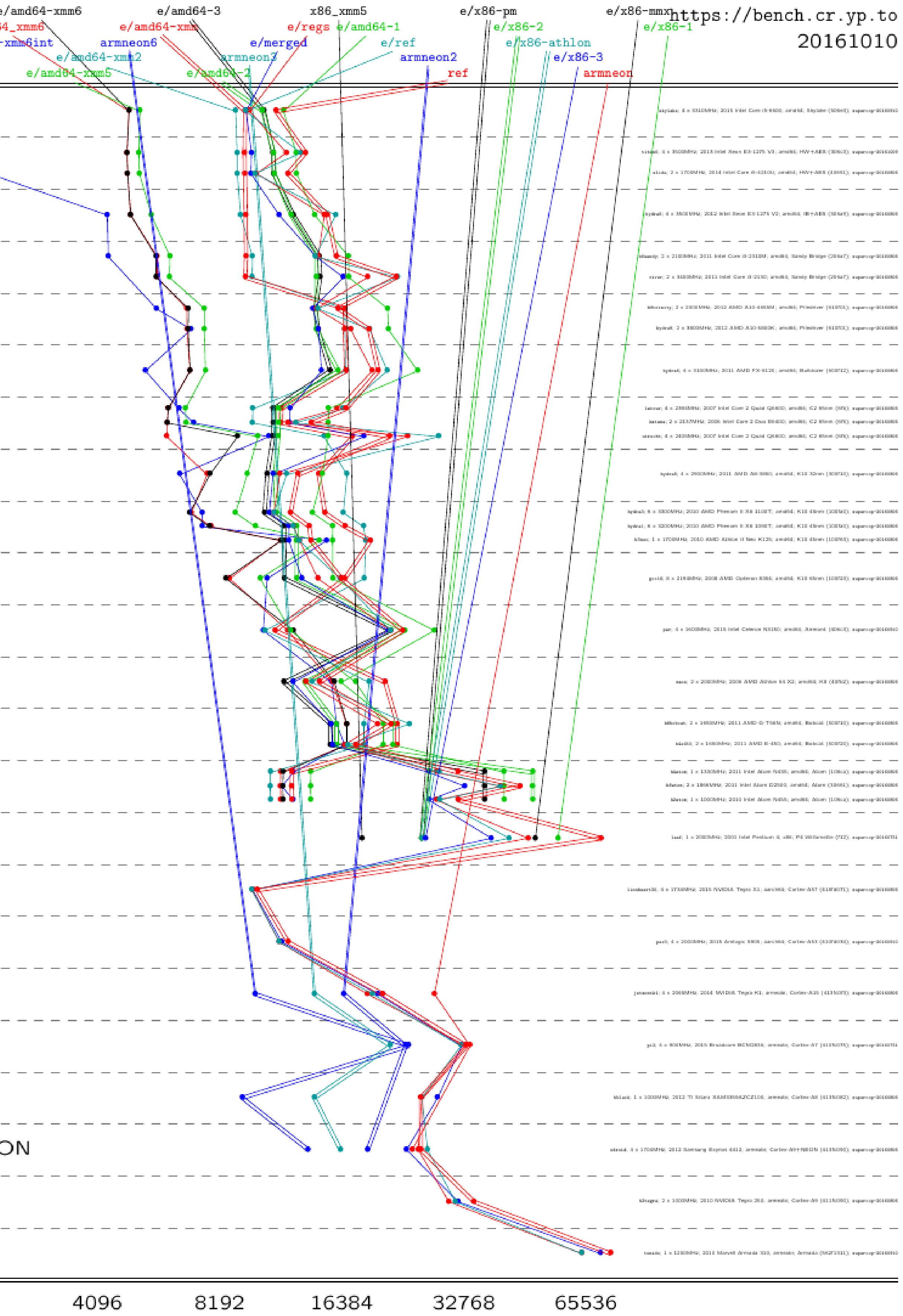


SUPERCOP benchmarking toolkit includes 2064 implementations of 563 cryptographic primitives. >20 implementations of Salsa20.

Haswell: Reasonably simple `ref` implementation compiled with `gcc -O3 -fomit-frame-pointer` is 6.15× slower than fastest Salsa20 implementation.

`merged` implementation with “machine-independent” optimizations and best of 121 compiler options: 4.52× slower.

s more complicated:



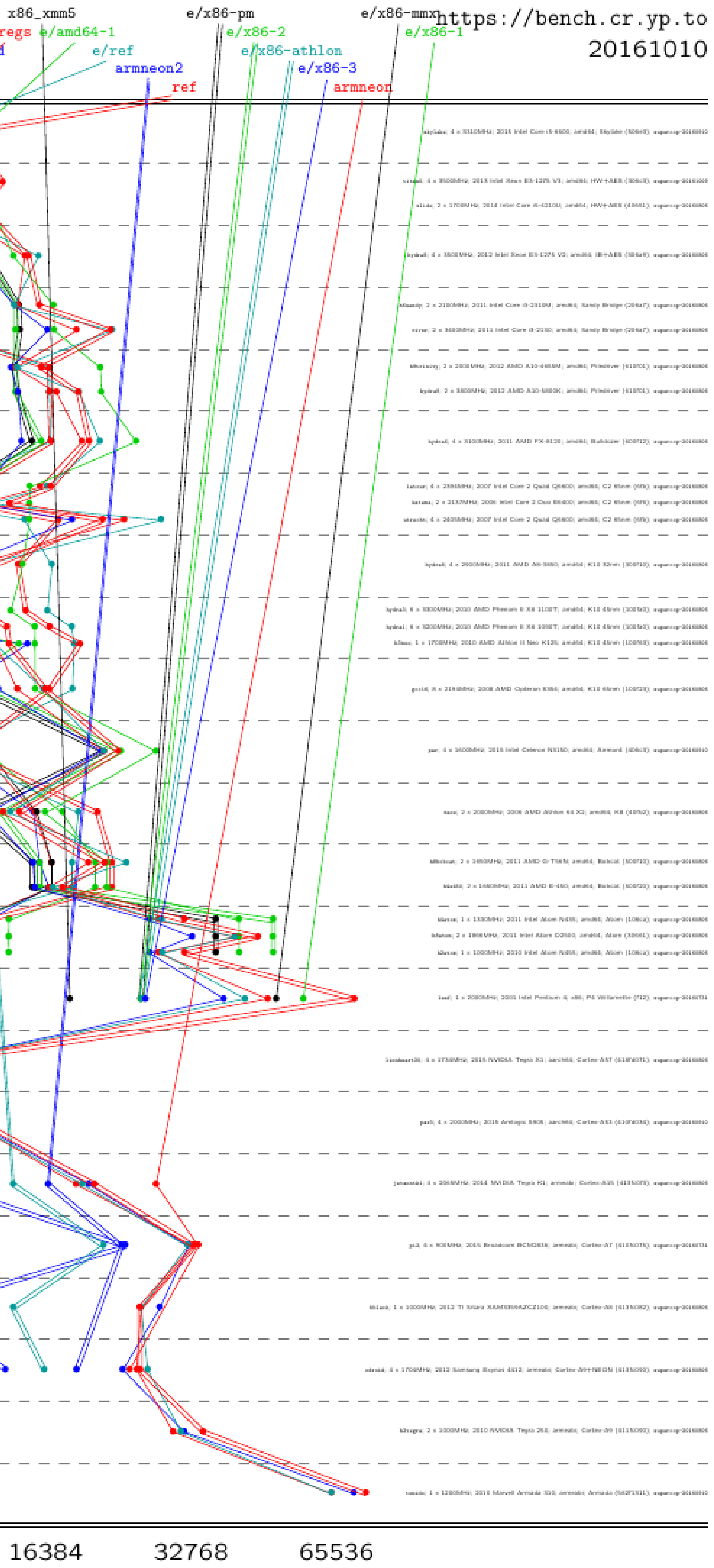
SUPERCOP benchmarking toolkit includes 2064 implementations of 563 cryptographic primitives. >20 implementations of Salsa20.

Haswell: Reasonably simple `ref` implementation compiled with `gcc -O3 -fomit-frame-pointer` is 6.15× slower than fastest Salsa20 implementation.

`merged` implementation with “machine-independent” optimizations and best of 121 compiler options: 4.52× slower.

Many m
were dev
to the (
impleme

mplicated:

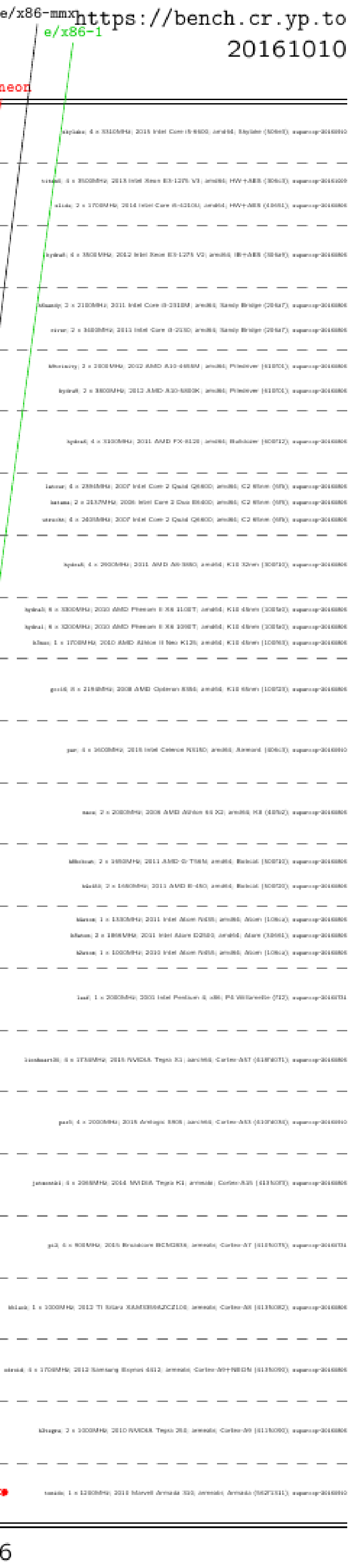


SUPERCOP benchmarking toolkit includes 2064 implementations of 563 cryptographic primitives. >20 implementations of Salsa20.

Haswell: Reasonably simple `ref` implementation compiled with `gcc -O3 -fomit-frame-pointer` is 6.15× slower than fastest Salsa20 implementation.

`merged` implementation with “machine-independent” optimizations and best of 121 compiler options: 4.52× slower.

Many more imple
were developed on
to the (currently)
implementation fo



SUPERCOP benchmarking toolkit includes 2064 implementations of 563 cryptographic primitives. >20 implementations of Salsa20.

Haswell: Reasonably simple `ref` implementation compiled with `gcc -O3 -fomit-frame-pointer` is 6.15× slower than fastest Salsa20 implementation.

`merged` implementation with "machine-independent" optimizations and best of 121 compiler options: 4.52× slower.

Many more implementations were developed on the way to the (currently) fastest implementation for this CPU

SUPERCOP benchmarking toolkit includes 2064 implementations of 563 cryptographic primitives. >20 implementations of Salsa20.

Haswell: Reasonably simple `ref` implementation compiled with `gcc -O3 -fomit-frame-pointer` is 6.15× slower than fastest Salsa20 implementation.

`merged` implementation with "machine-independent" optimizations and best of 121 compiler options: 4.52× slower.

Many more implementations were developed on the way to the (currently) fastest implementation for this CPU.

SUPERCOP benchmarking toolkit includes 2064 implementations of 563 cryptographic primitives. >20 implementations of Salsa20.

Haswell: Reasonably simple `ref` implementation compiled with `gcc -O3 -fomit-frame-pointer` is 6.15× slower than fastest Salsa20 implementation.

`merged` implementation with "machine-independent" optimizations and best of 121 compiler options: 4.52× slower.

Many more implementations were developed on the way to the (currently) fastest implementation for this CPU.

This is a common pattern. Very fast development cycle: modify the implementation, check that it still works, evaluate its performance.

SUPERCOP benchmarking toolkit includes 2064 implementations of 563 cryptographic primitives. >20 implementations of Salsa20.

Haswell: Reasonably simple `ref` implementation compiled with `gcc -O3 -fomit-frame-pointer` is 6.15× slower than fastest Salsa20 implementation.

`merged` implementation with "machine-independent" optimizations and best of 121 compiler options: 4.52× slower.

Many more implementations were developed on the way to the (currently) fastest implementation for this CPU.

This is a common pattern. Very fast development cycle: modify the implementation, check that it still works, evaluate its performance.

Results of each evaluation guide subsequent modifications.

SUPERCOP benchmarking toolkit includes 2064 implementations of 563 cryptographic primitives. >20 implementations of Salsa20.

Haswell: Reasonably simple `ref` implementation compiled with `gcc -O3 -fomit-frame-pointer` is 6.15× slower than fastest Salsa20 implementation.

`merged` implementation with "machine-independent" optimizations and best of 121 compiler options: 4.52× slower.

Many more implementations were developed on the way to the (currently) fastest implementation for this CPU.

This is a common pattern. Very fast development cycle: modify the implementation, check that it still works, evaluate its performance.

Results of each evaluation guide subsequent modifications.

The software engineer needs fast evaluation of performance.

COP benchmarking toolkit
2064 implementations
ryptographic primitives.
lementations of Salsa20.

Reasonably simple `ref`
ntation compiled with
`-fomit-frame-pointer`
slower than fastest
implementation.

implementation
achine-independent”
tions and best of 121
options: 4.52× slower.

Many more implementations
were developed on the way
to the (currently) fastest
implementation for this CPU.

This is a common pattern.
Very fast development cycle:
modify the implementation,
check that it still works,
evaluate its performance.

Results of each evaluation
guide subsequent modifications.

The software engineer needs fast evaluation of performance.

The unf

Slow eva
is often
to this o

nmarking toolkit
lementations
hic primitives.
ons of Salsa20.

bly simple `ref`
ompiled with
`frame-pointer`
an fastest
tation.

tation
lependent”
best of 121
4.52× slower.

Many more implementations were developed on the way to the (currently) fastest implementation for this CPU.

This is a common pattern. Very fast development cycle: modify the implementation, check that it still works, evaluate its performance.

Results of each evaluation guide subsequent modifications.

The software engineer needs fast evaluation of performance.

The unfortunate re

Slow evaluation of
is often a huge ob
to this optimizatio

toolkit
ons
ves.
sa20.

`ref`
th
`inter`

21
wer.

Many more implementations
were developed on the way
to the (currently) fastest
implementation for this CPU.

This is a common pattern.
Very fast development cycle:
modify the implementation,
check that it still works,
evaluate its performance.

Results of each evaluation
guide subsequent modifications.

The software engineer needs fast evaluation of performance.

The unfortunate reality:

Slow evaluation of performa
is often a huge obstacle
to this optimization process.

Many more implementations
were developed on the way
to the (currently) fastest
implementation for this CPU.

This is a common pattern.
Very fast development cycle:
modify the implementation,
check that it still works,
evaluate its performance.

Results of each evaluation
guide subsequent modifications.

The software engineer needs fast evaluation of performance.

The unfortunate reality:

Slow evaluation of performance
is often a huge obstacle
to this optimization process.

Many more implementations
were developed on the way
to the (currently) fastest
implementation for this CPU.

This is a common pattern.
Very fast development cycle:
modify the implementation,
check that it still works,
evaluate its performance.

Results of each evaluation
guide subsequent modifications.

The software engineer needs fast evaluation of performance.

The unfortunate reality:

Slow evaluation of performance
is often a huge obstacle
to this optimization process.

When performance evaluation is
too slow, the software engineer
has to switch context, and then
switching back to optimization
produces severe cache misses
inside software engineer’s brain.
(“I’m out of the zone.”)

Many more implementations
were developed on the way
to the (currently) fastest
implementation for this CPU.

This is a common pattern.
Very fast development cycle:
modify the implementation,
check that it still works,
evaluate its performance.

Results of each evaluation
guide subsequent modifications.

The software engineer needs fast evaluation of performance.

The unfortunate reality:

Slow evaluation of performance
is often a huge obstacle
to this optimization process.

When performance evaluation is
too slow, the software engineer
has to switch context, and then
switching back to optimization
produces severe cache misses
inside software engineer's brain.
("I'm out of the zone.")

Often optimization is aborted.
("I'll try some other time.")

ore implementations
veloped on the way
currently) fastest
ntation for this CPU.

common pattern.
t development cycle:
the implementation,
hat it still works,
its performance.

of each evaluation
bsequent modifications.

ftware engineer needs
luation of performance.

The unfortunate reality:

Slow evaluation of performance is often a huge obstacle to this optimization process.

When performance evaluation is too slow, the software engineer has to switch context, and then switching back to optimization produces severe cache misses inside software engineer's brain. ("I'm out of the zone.")

Often optimization is aborted. ("I'll try some other time.")

Goal of
Speed u
by speed

"Optimi
help opt

mentations
the way
fastest
r this CPU.

pattern.
ment cycle:
nentation,
works,
mance.

aluation
modifications.

**gineer needs
f performance.**

The unfortunate reality:

Slow evaluation of performance is often a huge obstacle to this optimization process.

When performance evaluation is too slow, the software engineer has to switch context, and then switching back to optimization produces severe cache misses inside software engineer's brain. ("I'm out of the zone.")

Often optimization is aborted. ("I'll try some other time.")

Goal of this talk:

Speed up the opti
by speeding up be

"Optimize benchm
help optimize opti

s

J.

:

ons.

eds
ance.

The unfortunate reality:

Slow evaluation of performance
is often a huge obstacle
to this optimization process.

When performance evaluation is
too slow, the software engineer
has to switch context, and then
switching back to optimization
produces severe cache misses
inside software engineer's brain.
("I'm out of the zone.")

Often optimization is aborted.
("I'll try some other time.")

Goal of this talk:

Speed up the optimization p
by speeding up benchmarkin

"Optimize benchmarking to
help optimize optimization."

The unfortunate reality:

Slow evaluation of performance is often a huge obstacle to this optimization process.

When performance evaluation is too slow, the software engineer has to switch context, and then switching back to optimization produces severe cache misses inside software engineer's brain. ("I'm out of the zone.")

Often optimization is aborted. ("I'll try some other time.")

Goal of this talk:
Speed up the optimization process by speeding up benchmarking.

"Optimize benchmarking to help optimize optimization."

The unfortunate reality:

Slow evaluation of performance is often a huge obstacle to this optimization process.

When performance evaluation is too slow, the software engineer has to switch context, and then switching back to optimization produces severe cache misses inside software engineer's brain. ("I'm out of the zone.")

Often optimization is aborted. ("I'll try some other time.")

Goal of this talk:
Speed up the optimization process by speeding up benchmarking.

"Optimize benchmarking to help optimize optimization."

What are the bottlenecks that really need speedups?
Measure the benchmarking process to gain understanding.

"Benchmark benchmarking to help optimize benchmarking."

ortunate reality:

aluation of performance
a huge obstacle
ptimization process.

erformance evaluation is
, the software engineer
witch context, and then
g back to optimization
s severe cache misses
ftware engineer's brain.
t of the zone.")

timization is aborted.
some other time.")

Goal of this talk:
Speed up the optimization process
by speeding up benchmarking.

"Optimize benchmarking to
help optimize optimization."

What are the bottlenecks
that really need speedups?
Measure the benchmarking
process to gain understanding.

"Benchmark benchmarking to
help optimize benchmarking."

Accessin

The soft
on his la
performa

eality:

f performance
stacle
on process.

e evaluation is
ware engineer
text, and then
optimization
ache misses
gineer's brain.
one.")

n is aborted.
er time.")

Goal of this talk:
Speed up the optimization process
by speeding up benchmarking.

"Optimize benchmarking to
help optimize optimization."

What are the bottlenecks
that really need speedups?
Measure the benchmarking
process to gain understanding.

"Benchmark benchmarking to
help optimize benchmarking."

Accessing different

The software engi
on his laptop, but
performance on m

nce

on is
eer
hen
ion
s
ain.

ed.

Goal of this talk:
Speed up the optimization process
by speeding up benchmarking.

“Optimize benchmarking to
help optimize optimization.”

What are the bottlenecks
that really need speedups?
Measure the benchmarking
process to gain understanding.

“Benchmark benchmarking to
help optimize benchmarking.”

Accessing different CPUs

The software engineer writes
on his laptop, but cares abo
performance on many more

Goal of this talk:
Speed up the optimization process by speeding up benchmarking.

“Optimize benchmarking to help optimize optimization.”

What are the bottlenecks that really need speedups?
Measure the benchmarking process to gain understanding.

“Benchmark benchmarking to help optimize benchmarking.”

Accessing different CPUs

The software engineer writes code on his laptop, but cares about performance on many more CPUs.

Goal of this talk:
Speed up the optimization process by speeding up benchmarking.

"Optimize benchmarking to help optimize optimization."

What are the bottlenecks that really need speedups?
Measure the benchmarking process to gain understanding.

"Benchmark benchmarking to help optimize benchmarking."

Accessing different CPUs

The software engineer writes code on his laptop, but cares about performance on many more CPUs.

Or at least *should* care!

Surprisingly common failure:
A paper with "faster algorithms" actually has slower algorithms running on faster processors.

Goal of this talk:
Speed up the optimization process by speeding up benchmarking.

“Optimize benchmarking to help optimize optimization.”

What are the bottlenecks that really need speedups?
Measure the benchmarking process to gain understanding.

“Benchmark benchmarking to help optimize benchmarking.”

Accessing different CPUs

The software engineer writes code on his laptop, but cares about performance on many more CPUs.

Or at least *should* care!

Surprisingly common failure:
A paper with “faster algorithms” actually has slower algorithms running on faster processors.

Systematic fix: Optimize each algorithm, new or old, for older and newer processors.

this talk:

p the optimization process
ding up benchmarking.

ze benchmarking to
imize optimization.”

e the bottlenecks
lly need speedups?
e the benchmarking
to gain understanding.

mark benchmarking to
imize benchmarking.”

Accessing different CPUs

The software engineer writes code on his laptop, but cares about performance on many more CPUs.

Or at least *should* care!

Surprisingly common failure:
A paper with “faster algorithms” actually has slower algorithms running on faster processors.

Systematic fix: Optimize each algorithm, new or old, for older and newer processors.

For each
Find a
copy co
(assumi
collect

mization process
nchmarking.

narking to
mization.”

lenecks
peedups?
nmarking
derstanding.

hmarking to
chmarking.”

Accessing different CPUs

The software engineer writes code on his laptop, but cares about performance on many more CPUs.

Or at least *should* care!

Surprisingly common failure:
A paper with “faster algorithms” actually has slower algorithms running on faster processors.

Systematic fix: Optimize each algorithm, new or old, for older and newer processors.

For each target C
Find a machine wi
copy code to that
(assuming it’s on
collect measureme

rocess
g.

ng.

to
.”

Accessing different CPUs

The software engineer writes code on his laptop, but cares about performance on many more CPUs.

Or at least *should* care!

Surprisingly common failure:
A paper with “faster algorithms” actually has slower algorithms running on faster processors.

Systematic fix: Optimize each algorithm, new or old, for older and newer processors.

For each target CPU:
Find a machine with that C
copy code to that machine
(assuming it’s on the Interne
collect measurements there.

Accessing different CPUs

The software engineer writes code on his laptop, but cares about performance on many more CPUs.

Or at least *should* care!

Surprisingly common failure:
A paper with "faster algorithms" actually has slower algorithms running on faster processors.

Systematic fix: Optimize
each algorithm, new or old,
for older and newer processors.

For each target CPU:
Find a machine with that CPU,
copy code to that machine
(assuming it's on the Internet),
collect measurements there.

Accessing different CPUs

The software engineer writes code on his laptop, but cares about performance on many more CPUs.

Or at least *should* care!

Surprisingly common failure:
A paper with "faster algorithms" actually has slower algorithms running on faster processors.

Systematic fix: Optimize
each algorithm, new or old,
for older and newer processors.

For each target CPU:
Find a machine with that CPU,
copy code to that machine
(assuming it's on the Internet),
collect measurements there.

But, for security reasons,
most machines on the Internet
disallow access by default,
except access by the owner.

Accessing different CPUs

The software engineer writes code on his laptop, but cares about performance on many more CPUs.

Or at least *should* care!

Surprisingly common failure:
A paper with "faster algorithms" actually has slower algorithms running on faster processors.

Systematic fix: Optimize each algorithm, new or old, for older and newer processors.

For each target CPU:
Find a machine with that CPU, copy code to that machine (assuming it's on the Internet), collect measurements there.

But, for security reasons, most machines on the Internet disallow access by default, except access by the owner.

Solution #1: Each software engineer buys each CPU.
This is expensive at high end, time-consuming at low end.

g different CPUs

tware engineer writes code
ptop, but cares about
ance on many more CPUs.

ast *should* care!

ngly common failure:
with “faster algorithms”
has slower algorithms
on faster processors.

tic fix: Optimize
orithm, new or old,
and newer processors.

For each target CPU:
Find a machine with that CPU,
copy code to that machine
(assuming it’s on the Internet),
collect measurements there.

But, for security reasons,
most machines on the Internet
disallow access by default,
except access by the owner.

Solution #1: Each software
engineer buys each CPU.
This is expensive at high end,
time-consuming at low end.

Solution

Poor cov

t CPUs

neer writes code
cares about
any more CPUs.

care!

non failure:
ter algorithms"
r algorithms
processors.

ptimize
ew or old,
er processors.

For each target CPU:
Find a machine with that CPU,
copy code to that machine
(assuming it's on the Internet),
collect measurements there.

But, for security reasons,
most machines on the Internet
disallow access by default,
except access by the owner.

Solution #1: Each software
engineer buys each CPU.
This is expensive at high end,
time-consuming at low end.

Solution #2: Ama
Poor coverage of

s code
ut
CPUs.

:
nms”
ns
.

ors.

For each target CPU:
Find a machine with that CPU,
copy code to that machine
(assuming it’s on the Internet),
collect measurements there.

But, for security reasons,
most machines on the Internet
disallow access by default,
except access by the owner.

Solution #1: Each software
engineer buys each CPU.
This is expensive at high end,
time-consuming at low end.

Solution #2: Amazon.
Poor coverage of CPUs.

For each target CPU:
Find a machine with that CPU,
copy code to that machine
(assuming it's on the Internet),
collect measurements there.

But, for security reasons,
most machines on the Internet
disallow access by default,
except access by the owner.

Solution #1: Each software
engineer buys each CPU.
This is expensive at high end,
time-consuming at low end.

Solution #2: Amazon.
Poor coverage of CPUs.

For each target CPU:
Find a machine with that CPU,
copy code to that machine
(assuming it's on the Internet),
collect measurements there.

But, for security reasons,
most machines on the Internet
disallow access by default,
except access by the owner.

Solution #1: Each software
engineer buys each CPU.
This is expensive at high end,
time-consuming at low end.

Solution #2: Amazon.
Poor coverage of CPUs.

Solution #3: Compile farms,
such as GCC Compile Farm.
Coverage of CPUs is better
but not good enough for crypto.
Usual goals are OS coverage
and architecture coverage.

For each target CPU:
Find a machine with that CPU, copy code to that machine (assuming it's on the Internet), collect measurements there.

But, for security reasons, most machines on the Internet disallow access by default, except access by the owner.

Solution #1: Each software engineer buys each CPU. This is expensive at high end, time-consuming at low end.

Solution #2: Amazon. Poor coverage of CPUs.

Solution #3: Compile farms, such as GCC Compile Farm. Coverage of CPUs is better but not good enough for crypto. Usual goals are OS coverage and architecture coverage.

Solution #4: Figure out who has the right machines. (How?) Send email saying "Are you willing to run this code?" Slow; unreliable; scales badly.

target CPU:
achine with that CPU,
de to that machine
g it's on the Internet),
easurements there.

security reasons,
chines on the Internet
access by default,
ccess by the owner.

#1: Each software
buys each CPU.
xpensive at high end,
suming at low end.

Solution #2: Amazon.
Poor coverage of CPUs.

Solution #3: Compile farms,
such as GCC Compile Farm.
Coverage of CPUs is better
but not good enough for crypto.
Usual goals are OS coverage
and architecture coverage.

Solution #4: Figure out who
has the right machines. (How?)
Send email saying "Are you
willing to run this code?"
Slow; unreliable; scales badly.

Solution
"Can I h
Saves ti

PU:
ith that CPU,
machine
the Internet),
nts there.

easons,
the Internet
default,
he owner.

n software
n CPU.
at high end,
t low end.

Solution #2: Amazon.
Poor coverage of CPUs.

Solution #3: Compile farms,
such as GCC Compile Farm.
Coverage of CPUs is better
but not good enough for crypto.
Usual goals are OS coverage
and architecture coverage.

Solution #4: Figure out who
has the right machines. (How?)
Send email saying “Are you
willing to run this code?”
Slow; unreliable; scales badly.

Solution #5: Sen
“Can I have an ac
Saves time but les

PU,

et),

net

d,

Solution #2: Amazon.
Poor coverage of CPUs.

Solution #3: Compile farms,
such as GCC Compile Farm.
Coverage of CPUs is better
but not good enough for crypto.
Usual goals are OS coverage
and architecture coverage.

Solution #4: Figure out who
has the right machines. (How?)
Send email saying “Are you
willing to run this code?”
Slow; unreliable; scales badly.

Solution #5: Send email say
“Can I have an account?”
Saves time but less reliable.

Solution #2: Amazon.
Poor coverage of CPUs.

Solution #3: Compile farms,
such as GCC Compile Farm.
Coverage of CPUs is better
but not good enough for crypto.
Usual goals are OS coverage
and architecture coverage.

Solution #4: Figure out who
has the right machines. (How?)
Send email saying "Are you
willing to run this code?"
Slow; unreliable; scales badly.

Solution #5: Send email saying
"Can I have an account?"
Saves time but less reliable.

Solution #2: Amazon.
Poor coverage of CPUs.

Solution #3: Compile farms,
such as GCC Compile Farm.
Coverage of CPUs is better
but not good enough for crypto.
Usual goals are OS coverage
and architecture coverage.

Solution #4: Figure out who
has the right machines. (How?)
Send email saying "Are you
willing to run this code?"
Slow; unreliable; scales badly.

Solution #5: Send email saying
"Can I have an account?"
Saves time but less reliable.

Solution #6: eBACS.

Good: One-time centralized
effort to find machines.

Solution #2: Amazon.
Poor coverage of CPUs.

Solution #3: Compile farms,
such as GCC Compile Farm.
Coverage of CPUs is better
but not good enough for crypto.
Usual goals are OS coverage
and architecture coverage.

Solution #4: Figure out who
has the right machines. (How?)
Send email saying "Are you
willing to run this code?"
Slow; unreliable; scales badly.

Solution #5: Send email saying
"Can I have an account?"
Saves time but less reliable.

Solution #6: eBACS.

Good: One-time centralized
effort to find machines.

Good: For each code submission,
one-time centralized audit.

Solution #2: Amazon.
Poor coverage of CPUs.

Solution #3: Compile farms,
such as GCC Compile Farm.
Coverage of CPUs is better
but not good enough for crypto.
Usual goals are OS coverage
and architecture coverage.

Solution #4: Figure out who
has the right machines. (How?)
Send email saying "Are you
willing to run this code?"
Slow; unreliable; scales badly.

Solution #5: Send email saying
"Can I have an account?"
Saves time but less reliable.

Solution #6: eBACS.

Good: One-time centralized
effort to find machines.

Good: For each code submission,
one-time centralized audit.

Good: High reliability,
high coverage, built-in tests.

Solution #2: Amazon.
Poor coverage of CPUs.

Solution #3: Compile farms,
such as GCC Compile Farm.
Coverage of CPUs is better
but not good enough for crypto.
Usual goals are OS coverage
and architecture coverage.

Solution #4: Figure out who
has the right machines. (How?)
Send email saying "Are you
willing to run this code?"
Slow; unreliable; scales badly.

Solution #5: Send email saying
"Can I have an account?"
Saves time but less reliable.

Solution #6: eBACS.

Good: One-time centralized
effort to find machines.

Good: For each code submission,
one-time centralized audit.

Good: High reliability,
high coverage, built-in tests.

Bad: Much too slow.

#2: Amazon.
verage of CPUs.

#3: Compile farms,
GCC Compile Farm.
e of CPUs is better
good enough for crypto.
oals are OS coverage
hitecture coverage.

#4: Figure out who
right machines. (How?)
nail saying “Are you
o run this code?”
nreliable; scales badly.

Solution #5: Send email saying “Can I have an account?” Saves time but less reliable.

Solution #6: eBACS.

Good: One-time centralized effort to find machines.

Good: For each code submission, one-time centralized audit.

Good: High reliability, high coverage, built-in tests.

Bad: Much too slow.

The eBA

Software
somethi

Software
sends pa
centraliz

eBACS
integrate

eBACS
new SUI
currently

azon.
CPUs.

npile farms,
pile Farm.
; is better
ugh for crypto.
S coverage
overage.

ure out who
hines. (How?)
"Are you
code?"
cales badly.

Solution #5: Send email saying "Can I have an account?"
Saves time but less reliable.

Solution #6: eBACS.

Good: One-time centralized effort to find machines.

Good: For each code submission, one-time centralized audit.

Good: High reliability, high coverage, built-in tests.

Bad: Much too slow.

The eBACS data f

Software engineer
something to benc

Software engineer
sends package by
centralized accoun

eBACS manager a
integrates into SU

eBACS manager b
new SUPERCOP
currently 26-mega

s,

ypto.

e

o

ow?)

y.

Solution #5: Send email saying
“Can I have an account?”
Saves time but less reliable.

Solution #6: eBACS.

Good: One-time centralized
effort to find machines.

Good: For each code submission,
one-time centralized audit.

Good: High reliability,
high coverage, built-in tests.

Bad: Much too slow.

The eBACS data flow

Software engineer has impl:
something to benchmark.

Software engineer submits i
sends package by email or (
centralized account) `git pu`

eBACS manager audits impl
integrates into SUPERCOP.

eBACS manager builds
new SUPERCOP package:
currently 26-megabyte `xz`.

Solution #5: Send email saying
“Can I have an account?”
Saves time but less reliable.

Solution #6: eBACS.

Good: One-time centralized effort to find machines.

Good: For each code submission, one-time centralized audit.

Good: High reliability, high coverage, built-in tests.

Bad: Much too slow.

The eBACS data flow

Software engineer has impl: something to benchmark.

Software engineer submits impl: sends package by email or (with centralized account) `git push`.

eBACS manager audits impl, integrates into SUPERCOP.

eBACS manager builds new SUPERCOP package: currently 26-megabyte `xz`.

#5: Send email saying
ave an account?”
me but less reliable.

#6: eBACS.

ne-time centralized
find machines.

or each code submission,
centralized audit.

igh reliability,
erage, built-in tests.

uch too slow.

The eBACS data flow

Software engineer has impl: something to benchmark.

Software engineer submits impl: sends package by email or (with centralized account) `git push`.

eBACS manager audits impl, integrates into SUPERCOP.

eBACS manager builds new SUPERCOP package: currently 26-megabyte `xz`.

eBACS
and ann

Each ma
waits un
is suffici

Each ma
downloa

SUPERC
stored o
On a typ
millions

d email saying
count?"
s reliable.

CS.

entralized
hines.

ode submission,
ed audit.

ility,
lt-in tests.

ow.

The eBACS data flow

Software engineer has impl: something to benchmark.

Software engineer submits impl: sends package by email or (with centralized account) `git push`.

eBACS manager audits impl, integrates into SUPERCOP.

eBACS manager builds new SUPERCOP package: currently 26-megabyte `xz`.

eBACS manager u
and announces pa

Each machine ope
waits until the ma
is sufficiently idle.

Each machine ope
downloads SUPER

SUPERCOP scans
stored on disk fro
On a typical high-
millions of files, se

ying

ssion,

.

The eBACS data flow

Software engineer has impl: something to benchmark.

Software engineer submits impl: sends package by email or (with centralized account) `git push`.

eBACS manager audits impl, integrates into SUPERCOP.

eBACS manager builds new SUPERCOP package: currently 26-megabyte `xz`.

eBACS manager uploads and announces package.

Each machine operator waits until the machine is sufficiently idle.

Each machine operator downloads SUPERCOP, run

SUPERCOP scans data stored on disk from previou
On a typical high-end CPU: millions of files, several GB.

The eBACS data flow

Software engineer has impl: something to benchmark.

Software engineer submits impl: sends package by email or (with centralized account) `git push`.

eBACS manager audits impl, integrates into SUPERCOP.

eBACS manager builds new SUPERCOP package: currently 26-megabyte `xz`.

eBACS manager uploads and announces package.

Each machine operator waits until the machine is sufficiently idle.

Each machine operator downloads SUPERCOP, runs it.

SUPERCOP scans data stored on disk from previous runs. On a typical high-end CPU: millions of files, several GB.

ACS data flow

e engineer has impl:
ng to benchmark.

e engineer submits impl:
ckage by email or (with
ed account) `git push`.

manager audits impl,
es into SUPERCOP.

manager builds
PERCOP package:
26-megabyte `xz`.

eBACS manager uploads and announces package.

Each machine operator waits until the machine is sufficiently idle.

Each machine operator downloads SUPERCOP, runs it.

SUPERCOP scans data stored on disk from previous runs. On a typical high-end CPU: millions of files, several GB.

For each
SUPERC

SUPERC
working
saves res

Typically

SUPERC
from thi
700-meg

Machine
`data.gz`

flow

has impl:
chmark.

submits impl:
email or (with
nt) `git push`.

udits impl,
PERCOP.

uilds
package:
byte `xz`.

eBACS manager uploads and announces package.

Each machine operator waits until the machine is sufficiently idle.

Each machine operator downloads SUPERCOP, runs it.

SUPERCOP scans data stored on disk from previous runs. On a typical high-end CPU: millions of files, several GB.

For each new impl
SUPERCOP comp

SUPERCOP meas
working compiled
saves results on di

Typically at least a

SUPERCOP collec
from this machine
700-megabyte dat

Machine operator
`data.gz`, announc

mpl:
with
`sh`.

,

eBACS manager uploads and announces package.

Each machine operator waits until the machine is sufficiently idle.

Each machine operator downloads SUPERCOP, runs it.

SUPERCOP scans data stored on disk from previous runs. On a typical high-end CPU: millions of files, several GB.

For each new impl-compiler
SUPERCOP compiles+tests

SUPERCOP measures each
working compiled impl,
saves results on disk.

Typically at least an hour.

SUPERCOP collects all data
from this machine, typically
700-megabyte `data.gz`.

Machine operator uploads
`data.gz`, announces it.

eBACS manager uploads
and announces package.

Each machine operator
waits until the machine
is sufficiently idle.

Each machine operator
downloads SUPERCOP, runs it.

SUPERCOP scans data
stored on disk from previous runs.
On a typical high-end CPU:
millions of files, several GB.

For each new impl-compiler pair,
SUPERCOP compiles+tests impl.

SUPERCOP measures each
working compiled impl,
saves results on disk.

Typically at least an hour.

SUPERCOP collects all data
from this machine, typically
700-megabyte `data.gz`.

Machine operator uploads
`data.gz`, announces it.

manager uploads
ounces package.

achine operator
til the machine
ently idle.

achine operator
ds SUPERCOP, runs it.

COP scans data
n disk from previous runs.
pical high-end CPU:
of files, several GB.

For each new impl-compiler pair, SUPERCOP compiles+tests impl.

SUPERCOP measures each working compiled impl, saves results on disk.

Typically at least an hour.

SUPERCOP collects all data from this machine, typically 700-megabyte `data.gz`.

Machine operator uploads `data.gz`, announces it.

eBACS
`data.gz`

Databas
53% cur
47% arc

For each
(or for c
scripts p
Typically

Web pag
Under ar

uploads

ckage.

erator

chine

erator

RCOP, runs it.

s data

m previous runs.

end CPU:

everal GB.

For each new impl-compiler pair, SUPERCOP compiles+tests impl.

SUPERCOP measures each working compiled impl, saves results on disk.

Typically at least an hour.

SUPERCOP collects all data from this machine, typically 700-megabyte `data.gz`.

Machine operator uploads `data.gz`, announces it.

eBACS manager c

`data.gz` into cent

Database currently

53% current unco

47% archives of su

For each new `data`

(or for cross-cuttin

scripts process all

Typically an hour

Web pages are reg

Under an hour.

s it.

s runs.

For each new impl-compiler pair, SUPERCOP compiles+tests impl.

SUPERCOP measures each working compiled impl, saves results on disk.

Typically at least an hour.

SUPERCOP collects all data from this machine, typically 700-megabyte `data.gz`.

Machine operator uploads `data.gz`, announces it.

eBACS manager copies `data.gz` into central databa

Database currently uses 500
53% current uncompressed
47% archives of superseded

For each new `data.gz`
(or for cross-cutting update
scripts process all results.
Typically an hour per machi

Web pages are regenerated.
Under an hour.

For each new impl-compiler pair, SUPERCOP compiles+tests impl.

SUPERCOP measures each working compiled impl, saves results on disk.

Typically at least an hour.

SUPERCOP collects all data from this machine, typically 700-megabyte `data.gz`.

Machine operator uploads `data.gz`, announces it.

eBACS manager copies `data.gz` into central database.

Database currently uses 500GB: 53% current uncompressed data, 47% archives of superseded data.

For each new `data.gz` (or for cross-cutting updates): scripts process all results. Typically an hour per machine.

Web pages are regenerated. Under an hour.

new impl-compiler pair,
COP compiles+tests impl.

COP measures each
compiled impl,
sults on disk.

at least an hour.

COP collects all data
s machine, typically
gabyte `data.gz`.

operator uploads
, announces it.

eBACS manager copies
`data.gz` into central database.

Database currently uses 500GB:
53% current uncompressed data,
47% archives of superseded data.

For each new `data.gz`
(or for cross-cutting updates):
scripts process all results.
Typically an hour per machine.

Web pages are regenerated.
Under an hour.

In progre

New dat

All impls

Some m
measure
“publish
does for

All comp

All chec

All meas

All table

-compiler pair,
piles+tests impl.

ures each
impl,
sk.

an hour.

cts all data
, typically
a.gz.

uploads
ces it.

---

eBACS manager copies
`data.gz` into central database.

Database currently uses 500GB:
53% current uncompressed data,
47% archives of superseded data.

For each new `data.gz`
(or for cross-cutting updates):
scripts process all results.
Typically an hour per machine.

Web pages are regenerated.
Under an hour.

In progress: SUPE

New database stor

All impls ever sub

Some metadata no
measurements. Bu
“publish results” f
does force new me

All compiled impls

All checksums of

All measurements.

All tables, graphs,

pair,
impl.

a

eBACS manager copies `data.gz` into central database.

Database currently uses 500GB:
53% current uncompressed data,
47% archives of superseded data.

For each new `data.gz`
(or for cross-cutting updates):
scripts process all results.
Typically an hour per machine.

Web pages are regenerated.
Under an hour.

In progress: SUPERCOP 2

New database stored centra

All impls ever submitted.

Some metadata not affectin
measurements. But turning
"publish results" for an imp
does force new measuremen

All compiled impls.

All checksums of outputs.

All measurements.

All tables, graphs, etc.

eBACS manager copies `data.gz` into central database.

Database currently uses 500GB: 53% current uncompressed data, 47% archives of superseded data.

For each new `data.gz` (or for cross-cutting updates): scripts process all results. Typically an hour per machine.

Web pages are regenerated. Under an hour.

In progress: SUPERCOP 2

New database stored centrally:

All impls ever submitted.

Some metadata not affecting measurements. But turning on "publish results" for an impl *does* force new measurements.

All compiled impls.

All checksums of outputs.

All measurements.

All tables, graphs, etc.

manager copies
z into central database.

e currently uses 500GB:
rrent uncompressed data,
hives of superseded data.

n new `data.gz`
ross-cutting updates):
rocess all results.
y an hour per machine.

ges are regenerated.
n hour.

In progress: SUPERCOP 2

New database stored centrally:

All impls ever submitted.

Some metadata not affecting measurements. But turning on “publish results” for an impl *does* force new measurements.

All compiled impls.

All checksums of outputs.

All measurements.

All tables, graphs, etc.

When ne

Impl is p

Each co
to check

Each wo
pushed t
(when th

Each me
immedia

If impl s
Measure
after co

opies

tral database.

y uses 500GB:

mpressed data,

uperseded data.

`a.gz`

ng updates):

results.

per machine.

generated.

In progress: SUPERCOP 2

New database stored centrally:

All impls ever submitted.

Some metadata not affecting measurements. But turning on “publish results” for an impl *does* force new measurements.

All compiled impls.

All checksums of outputs.

All measurements.

All tables, graphs, etc.

When new impl is

Impl is pushed to

Each compiled imp

to checksum mach

Each working com

pushed to benchm

(when they are su

Each measuremen

immediately to sul

If impl says “publi

Measurements are

after comparisons

ase.

GB:

data,

data.

s):

ne.

In progress: SUPERCOP 2

New database stored centrally:

All impls ever submitted.

Some metadata not affecting measurements. But turning on “publish results” for an impl *does* force new measurements.

All compiled impls.

All checksums of outputs.

All measurements.

All tables, graphs, etc.

When new impl is submitte

Impl is pushed to compile se

Each compiled impl is pushe
to checksum machines.

Each working compiled impl
pushed to benchmark machi
(when they are sufficiently i

Each measurement is availa
immediately to submitter.

If impl says “publish results”
Measurements are put onlin
after comparisons are done.

In progress: SUPERCOP 2

New database stored centrally:

All impls ever submitted.

Some metadata not affecting measurements. But turning on "publish results" for an impl *does* force new measurements.

All compiled impls.

All checksums of outputs.

All measurements.

All tables, graphs, etc.

When new impl is submitted:

Impl is pushed to compile servers.

Each compiled impl is pushed to checksum machines.

Each working compiled impl is pushed to benchmark machines (when they are sufficiently idle).

Each measurement is available immediately to submitter.

If impl says "publish results":
Measurements are put online after comparisons are done.

ess: SUPERCOP 2

tabase stored centrally:

s ever submitted.

etadata not affecting
ments. But turning on
results” for an impl
ce new measurements.

piled impls.

ksums of outputs.

surements.

es, graphs, etc.

When new impl is submitted:

Impl is pushed to compile servers.

Each compiled impl is pushed to checksum machines.

Each working compiled impl is pushed to benchmark machines (when they are sufficiently idle).

Each measurement is available immediately to submitter.

If impl says “publish results”: Measurements are put online after comparisons are done.

Wait, wl

No more
there’s r

Critical i
Can a ro
take ove
Or corru
from oth

ERCOP 2

red centrally:

mitted.

ot affecting
ut turning on
or an impl
easurements.

s.

outputs.

etc.

When new impl is submitted:

Impl is pushed to compile servers.

Each compiled impl is pushed
to checksum machines.

Each working compiled impl is
pushed to benchmark machines
(when they are sufficiently idle).

Each measurement is available
immediately to submitter.

If impl says “publish results”:
Measurements are put online
after comparisons are done.

Wait, what about

No more central a
there’s no time for

Critical integrity c
Can a rogue code
take over the mac
Or corrupt benchn
from other submit

lly:

g

on

l

ts.

When new impl is submitted:

Impl is pushed to compile servers.

Each compiled impl is pushed to checksum machines.

Each working compiled impl is pushed to benchmark machines (when they are sufficiently idle).

Each measurement is available immediately to submitter.

If impl says “publish results”: Measurements are put online after comparisons are done.

Wait, what about security?

No more central auditing: there’s no time for it.

Critical integrity concerns:
Can a rogue code submitter take over the machine?
Or corrupt benchmarks from other submitters?

When new impl is submitted:

Impl is pushed to compile servers.

Each compiled impl is pushed
to checksum machines.

Each working compiled impl is
pushed to benchmark machines
(when they are sufficiently idle).

Each measurement is available
immediately to submitter.

If impl says "publish results":
Measurements are put online
after comparisons are done.

Wait, what about security?

No more central auditing:
there's no time for it.

Critical integrity concerns:
Can a rogue code submitter
take over the machine?
Or corrupt benchmarks
from other submitters?

When new impl is submitted:

Impl is pushed to compile servers.

Each compiled impl is pushed
to checksum machines.

Each working compiled impl is
pushed to benchmark machines
(when they are sufficiently idle).

Each measurement is available
immediately to submitter.

If impl says "publish results":
Measurements are put online
after comparisons are done.

Wait, what about security?

No more central auditing:
there's no time for it.

Critical integrity concerns:
Can a rogue code submitter
take over the machine?
Or corrupt benchmarks
from other submitters?

Concerns start before code is
tested and measured: compilers
have bugs, sometimes serious.

When new impl is submitted:

Impl is pushed to compile servers.

Each compiled impl is pushed
to checksum machines.

Each working compiled impl is
pushed to benchmark machines
(when they are sufficiently idle).

Each measurement is available
immediately to submitter.

If impl says "publish results":
Measurements are put online
after comparisons are done.

Wait, what about security?

No more central auditing:
there's no time for it.

Critical integrity concerns:
Can a rogue code submitter
take over the machine?
Or corrupt benchmarks
from other submitters?

Concerns start before code is
tested and measured: compilers
have bugs, sometimes serious.

Smaller availability concerns:
e.g., Bitcoin mining.

ew impl is submitted:

ushed to compile servers.

mpiled impl is pushed
ksum machines.

rking compiled impl is
to benchmark machines
hey are sufficiently idle).

easurement is available
tely to submitter.

ays “publish results”:
ements are put online
mparisons are done.

Wait, what about security?

No more central auditing:
there’s no time for it.

Critical integrity concerns:
Can a rogue code submitter
take over the machine?
Or corrupt benchmarks
from other submitters?

Concerns start before code is
tested and measured: compilers
have bugs, sometimes serious.

Smaller availability concerns:
e.g., Bitcoin mining.

SUPER
OS-level
impl can
cannot f

SUPER
pool of
each co
machine

Enforces
for files
in comp

More dif
integrity
tables co

submitted:

compile servers.

pl is pushed
nines.

piled impl is
ark machines
fficiently idle).

t is available
bmitter.

sh results":
put online
are done.

Wait, what about security?

No more central auditing:
there's no time for it.

Critical integrity concerns:
Can a rogue code submitter
take over the machine?
Or corrupt benchmarks
from other submitters?

Concerns start before code is
tested and measured: compilers
have bugs, sometimes serious.

Smaller availability concerns:
e.g., Bitcoin mining.

SUPERCOP 1 sets
OS-level resource
impl cannot open
cannot fork any pr

SUPERCOP 2 ma
pool of uids and c
each compile serve
machine, benchma

Enforces reasonab
for files legitimatel
in compiling an im

More difficult to e
integrity policy for
tables comparing i

d:

ervers.

ed

is

nes

dle).

ple

:

e

Wait, what about security?

No more central auditing: there's no time for it.

Critical integrity concerns: Can a rogue code submitter take over the machine? Or corrupt benchmarks from other submitters?

Concerns start before code is tested and measured: compilers have bugs, sometimes serious.

Smaller availability concerns: e.g., Bitcoin mining.

SUPERCOP 1 sets some OS-level resource limits: impl cannot open any files, cannot fork any processes.

SUPERCOP 2 manages pool of uids and chroot jails
each compile server, checksu
machine, benchmark machin

Enforces reasonable policy for files legitimately used in compiling an impl.

More difficult to enforce: integrity policy for, e.g., tables comparing impls.

Wait, what about security?

No more central auditing: there's no time for it.

Critical integrity concerns: Can a rogue code submitter take over the machine? Or corrupt benchmarks from other submitters?

Concerns start before code is tested and measured: compilers have bugs, sometimes serious.

Smaller availability concerns: e.g., Bitcoin mining.

SUPERCOP 1 sets some OS-level resource limits: impl cannot open any files, cannot fork any processes.

SUPERCOP 2 manages pool of uids and chroot jails on each compile server, checksum machine, benchmark machine.

Enforces reasonable policy for files legitimately used in compiling an impl.

More difficult to enforce: integrity policy for, e.g., tables comparing impls.